Sub A1

Methods and Apparatus for Improving Fetching and Dispatch of Instructions in Multithreaded Processors

by inventors Enric Musoll and Mario Nemirovsky

Field of the Invention

The present invention is in the area of microprocessors, and pertains more particularly to structure and function of simultaneous multithreaded processors.

Cross Reference to Related Documents

The present application is a continuation-in-part (CIP) of prior co-pending patent applications 09/216,017, filed 12/16/98, 09/240,012, filed 1/27/99, 09/273,810, filed 3/22/99 and 09/312,302 filed 5/14/99 all four of which are incorporated herein in their entirety by reference.

Background of the Invention

Sub A2

Multi-streaming processors capable of processing multiple threads are known in the art, and have been the subject of considerable research and development. The present invention takes notice of the prior work in this field, and builds upon that work, bringing new and non-obvious improvements in apparatus and methods to the art. The inventors have provided with this patent application an Information Disclosure Statement listing a number of published papers in the technical field of multi-streaming processors, which together provide additional background and context for the several aspects of the present invention disclosed herein.

For purposes of definition, this specification regards a *stream* in reference to a processing system as a *hardware* capability of the processor for supporting and processing an instruction thread. A *thread* is the actual software running within a stream. For example, a multi-streaming processor implemented as a CPU for operating a desktop computer may simultaneously process threads from two or more applications, such as a word processing program and an object-oriented drawing program. As another example, a multi-streaming-capable processor may operate a machine without regular human direction, such as a router in a packet switched network. In a router, for example, there may be one or more threads for processing and forwarding data packets on the network, another for quality-of-service (QoS) negotiation with other routers and servers connected to the network and another for maintaining routing tables and the like. The maximum capability of any multi-streaming processor to process multiple concurrent *threads* remains fixed at the number of hardware *streams* the processor supports.

A multi-streaming processor operating a single thread runs as a single-stream processor with unused streams idle. For purposes of discussion, a stream is considered an *active* stream at all times the stream supports a thread, and otherwise inactive. As in various related cases listed under the cross-reference section, and in papers provided by IDS, which were included with at least one of the cross-referenced applications, superscalar processors are also known in the art. This term refers to processors that have multiples of one or more types of functional units, and an ability to issue concurrent instructions to multiple functional units. Most central processing units (CPUs) built today have more than a single functional unit of each type, and are thus superscalar processors by this definition. Some have many such units, including, for example, multiple

floating point units, integer units, logic units, load/store units and so forth. Multi-streaming superscalar processors are known in the art as well.

State-of-the-art processors typically employ pipelining, whether the processor is a single streaming processor, or a dynamic multi-streaming processor. As is known in the art, pipelining is a technique in which multiple instructions are queued in steps leading to execution, thus speeding up instruction execution. Most processors pipeline instruction execution, so instructions take several steps until they are executed. A brief description of typical stages in a RISC architecture is listed immediately below:

a) Fetch stage: instructions are fetched from memory
b) Decode stage: instructions are decoded
c) Read/Dispatch stage: source operands are read from register file
d) Execute stage: operations are executed, an address is calculated or a branch is resolved
e) Access stage: data is accessed
f) Write stage: the result is written in a register

Pipeline stages take a single clock cycle, so the cycle must be long enough to allow for the slowest operation. The present invention is related the fact that there are situations in pipelining when instructions cannot be executed. Such events are called hazards in the art. Commonly, there are three types of hazards:

a) Structural
b) Data
c) Control

A structural hazard means that there are not adequate resources (e.g., functional units) to support the combination of instructions to be executed in the same clock cycle. A data hazard arises when an instruction depends on the result of one or more previous instructions not resolved. Forwarding or bypassing techniques are commonly used to reduce the

impact of data hazards. A control hazard arises from the pipelining of branches and other instructions that change the program counter (PC). In this case the pipeline may be stalled until the branch is resolved.

Stalling on branches has a dramatic impact onto processor performance (measured in instructions executed per cycle or IPC). The longer the pipelines and the wider the superscalar, the more substantial is the negative impact. Since the cost of stalls is quite high, it is common in the art to predict the outcome of branches. Branch predictors predict branches as either *taken* or *untaken* and the target address. Branch predictors may be either static or dynamic. Dynamic branch predictors may change prediction for a given branch during program execution.

A typical approach to branch prediction is to keep a history for each branch, and then to use the past to predict the future. For example, if a given branch has always been taken in the past, there is a high probability that the same branch will be taken again in the future. On the other hand, if the branch was taken 2 times, not taken 5 times, taken again once, and so forth, the prediction made will have a low confidence level. When the prediction is wrong, the pipeline must be flushed, and the pipeline control must ensure that the instructions following the wrongly guessed branch are discarded, and must restart the pipeline from the proper target address. This is a costly operation.

Multistreaming processor architectures may be either fine-grained or coarse-grained. Coarse-grained multistreaming processors typically have multiple contexts, which are used to cover long latencies arising, for example, due to cache misses. Only a single thread is executing at a given time. In contrast, fine-grained multistreaming technologies such as Dynamic Multi-Streaming (DMS), which is a development of XStream Logic, Inc.,

with which the present inventors are associated, allow true multi-tasking or multistreaming in a single processor, concurrently executing instructions from multiple distinct threads or tasks. DMS processors implement multiple sets of CPU registers or hardware contexts to support this style of execution.

Increasing the relative amount of instruction level parallelism (ILP) for a processor reduces data and control hazards, so applications can exploit increasing number of functional units during peak levels of parallelism, and Dynamic Multi-Streaming (DMS) hardware and techniques within today's general-purpose superscalar processors significantly improves performance by increasing the amount of ILP, and more evenly distributing it within workload. There are still occasions, however, for degraded performance due to poor selection in fetching and dispatching instructions in a DMS processor.

What is clearly needed is improved methods and apparatus for utilizing hit/miss prediction in pipelines in dynamic multi-streaming processors, particularly at the point of fetch and dispatch operations.

Summary of the Invention

In a preferred embodiment of the present invention, in a multi-streaming processor having a data cache, a system for fetching instructions from individual ones of the multiple streams to a pipeline is provided, comprising a fetch algorithm for selecting from which stream to fetch instructions, and a hit/miss predictor for forecasting whether instructions will hit or miss the data cache. The prediction by the hit-miss predictor is used by the fetch algorithm in determining from which stream to fetch.

In preferred embodiments a hit prediction precipitates no change in the fetching process, while a miss prediction results in switching fetching to a different stream. In some cases the hit-miss predictor determines a hit probability, and the probability is used by the fetch algorithm in determining from where to fetch next instructions. In some other embodiments the forecast of the hit/miss predictor is also used by a dispatch algorithm in selecting instructions from the pipeline to dispatch to functional units.

In another aspect of the invention a multi-streaming processor is provided, comprising a data cache, a fetch algorithm for selecting from which stream to fetch instructions, and a hit/miss predictor for predicting whether instructions will hit or miss the cache. A prediction by the hit-miss predictor is used by the fetch algorithm in determining from which stream to fetch.

In preferred embodiments of the invention a hit prediction precipitates no change in the fetching process, while a miss prediction results in switching fetching to a different stream. In some embodiments the hit-miss predictor determines a hit probability, and the probability is used by the fetch algorithm in determining from where to fetch instructions.

In some embodiments the forecast of the hit/miss predictor is also used by a dispatch algorithm in selecting instructions from the pipeline to dispatch to functional units.

In yet another aspect of the invention, in a multi-streaming processor having a data cache, a method for fetching instructions from individual ones of multiple streams as instruction sources to a pipeline is provided, comprising the steps of (a) making a hit/miss prediction by a predictor as to whether instructions previously fetched will hit or miss the data cache; and (b) if the prediction is a miss, altering the source of the fetch. In some embodiments the hit-miss predictor determines a hit probability, and the

probability is used in determining fetch source. In other embodiments the forecast of the hit/miss predictor is also used by a dispatch algorithm in selecting instructions to dispatch to functional units.

In embodiments of the invention taught in enabling detail below, for the first time a prediction technique is brought to the process of fetching and dispatching instruction in multistreaming processors.

Brief Description of the Drawing Figures

Fig. 1a is a simplified diagram of a pipeline in an embodiment of the present invention.

Fig. 1b shows the pipeline of Fig. 1a after a cycle.

Fig. 1c shows the pipeline of Fig. 1a and 1b after another cycle.

Fig. 1d shows the pipeline of Fig. 1a, 1b and 1c after yet another cycle.

Fig. 2 is a schematic diagram associating predictors with streams in an embodiment of the present invention.

Fig. 3 is a schematic showing predictors for different levels in cache.

Fig. 4 is a schematic illustrating benefits of the technique in embodiments of the invention.

Description of the Preferred Embodiments

Fig. 1a is a simplified diagram of a pipeline in a dynamic, multi-streaming (DMS) processor according to an embodiment of the present

Cont A4

invention. In this simplified view the pipeline has seven stages, which are fetch, decode, read, dispatch, execute, access and write. These are the same as described in the background section above, except for the separation of read and dispatch in Fig. 1a to illustrate the functions. Dispatch is important in the present invention in that the present invention adds intelligence to Dispatch, improving the performance of the processor. The fetch stage in the pipeline fetches instructions into the pipeline from the multiple streams, and in an embodiment of the present invention is capable of selective fetching.

Sub A5

Although there is no requirement in operating processors that there be instructions at each stage of a pipeline, it is often true that this is the case, and the inventors choose to illustrate each stage as occupied by a single instruction to avoid confusion in description. In many cases there will a plurality of instructions at various stages, or none at all.

In Fig. 1a the instructions in the pipeline are arbitrarily indicated as instructions A through G, at successive stages in the pipeline at one point in time. Fig. 1b shows the pipeline of Fig. 1a one cycle later. Note that instruction A has moved from fetch to decode, and the other instructions shown in Fig. 1a have moved one stage forward as well. Also, a new instruction, H, has entered the pipeline at the fetch stage.

Fig. 1c shows the same pipeline one cycle later. All instructions have moved forward one further stage, and a new instruction I has entered the pipeline at the fetch stage. Fig. 1d shows the same pipeline after yet another cycle, at which point in time the instructions have moved forward yet again, and yet another instruction J has entered the pipeline.

Note that after the fourth cycle, instruction A has moved from fetch to dispatch. Assume for the sake of this example that instruction A is a load instruction for loading a data value from cache. If this is the case, there will

be some probability as to whether the particular data is in cache or not. In the art this is known as the hit/miss probability. If the data is in the cache, the system scores a hit. If not, the system scores a miss.

The combination of hit/miss probability for load operations with pipelined architecture has significance for processor efficiency, because, in the conventional case the general sequence of instructions in the pipeline will be from a single thread, and will typically be related in that many instructions following a load instruction may depend upon the result of whatever instruction is to use the data loaded. That is, until the resolution of whatever instruction is to use the data loaded, many following instructions cannot be executed, except in some cases, on a speculative basis.

Conventional processors simply assume a hit when a load instruction enters a pipeline. If the load is a *miss*, however, once the load instruction is executed, then it may take a number of cycles for the needed data, not in cache, to be loaded from memory. And, unfortunately, the miss will not be apparent until the load instruction is dispatched and executed. The following instructions have to stall until the data is loaded and the instruction(s) depending on the data are executed.

The present inventors provide apparatus and methods for reducing the impact of data cache misses in multithreaded architectures. The technique consists of predicting, for each of the threads running in the multiple streams of the DMS, whether the next access to the data cache will result in a miss. If this is the case, then (generally):

- The stream can be given a lower priority when deciding, in the fetch stage, from which stream to fetch, and
- The dependent instructions of the instruction that accesses the data cache can be more efficiently dispatched to the functional units (FU's) in the dispatch stage.

This new apparatus and technique improves the performance of a multistreaming processor in the fetching and dispatching of instructions.

Fetch

The new technique takes advantage of the fact that, in DMS processor, as instructions are fetched to the pipeline from individual ones of the streams, there is freedom in choosing a fetching policy or algorithm that will select, on a cycle-by-cycle basis, from which stream instructions are to be fetched.

In a multistreaming architecture, without the technique proposed here, a typical event that causes a thread switch is a data cache miss. Since the required data may take several cycles to be available (the exact number depending on where the data really resides in the memory hierarchy of the processor), the thread that missed the data cache may be switched out since the dependent instructions of the instruction that missed most likely will not execute due to the dependencies on the data. Thus, more work can be done by fetching and executing instructions from another thread. In this case, the instructions following the one that missed, and that have already been fetched, will need to be flushed out, thus degrading the performance of the processor with respect to the case in which useful instructions had been fetched.

If the fact that an instruction will miss the data cache could be known early in the process the fetching of instructions that might eventually be flushed may be avoided by fetching, instead of the instructions following the instruction that missed the data cache, instructions from another stream, improving the likelihood that the fetched instructions may be quickly

executed. Thus, a fetching algorithm, in an embodiment of the present invention, may take into account, for all the streams, the predictions on whether the next access will miss the data cache, and fetch from the stream running a thread that is most likely to have its instructions executed and committed.

There already exist in the art a variety of implementations for hit-miss predictors. The goal, however, is always the same: to predict with the highest accuracy both the hits and misses to the data cache. Moreover, a desirable property of such a predictor is to be able to predict the next access to the data cache as soon as possible so that fewer instructions (that would eventually be flushed out) will enter the pipeline.

The technique taught herein can be improved by associating a confidence level to the prediction. The predictor, in one embodiment of the invention, operating at the fetch stage, in addition to predicting also generates this confidence level value. The confidence level helps the fetching algorithm, for example, in cases in which two or more predictors predicted a miss in the data cache and one is selected to be switched out. In this case, the stream with higher confidence level will be selected.

Fig. 2 is a schematic diagram of a fetching algorithm in a multistreaming architecture. The algorithm decides from which stream(s) to fetch based on cache hit/miss predictors associated to each of the streams. In Fig. 2 a predictor is associated with streams 1, 2, and so on through stream S. Thus, theoretically, instructions from up to S streams (S being the maximum number of streams supported by the multistreaming architecture) can be simultaneously fetched every cycle. In reality, however, the fetching algorithm might be restricted to fetch instructions from P streams ($P < S$) due to implementation restrictions (for example, availability of instruction cache ports). Moreover, the fetching algorithm might select from which

streams to fetch based on other information (for example, confidence on the branch prediction for each stream, thread priorities, state of the pipeline, etc.)

So far, we have mentioned predictors of hit/miss for the data cache. Note that the data cache might be implemented for performance reasons in different levels (the first level - L1- being the closest to the processor core). In alternative embodiments of the invention different hit/miss predictors may exist for each of the data cache levels.

The fetching algorithm in alternative embodiments of the present invention may base selection of instructions to be fetched on the prediction for the second level - L2 - of data cache since, in most processor systems, a miss in the second level of cache is very costly in number of cycles (whereas the penalty of a miss in the L1 is comparatively relatively small).

Dispatch

The technique of having a data cache hit/miss predictor is also useful in the process of deciding, at the dispatch stage in the pipeline, which instructions are to be extracted from the instruction queue (if any) and sent to the functional units (FUs) for execution.

In current art, when an instruction (henceforth called a producer) generates a read access to the data cache, the latency of the result is not known until the data cache is accessed and the hit/miss outcome is determined. The dispatch of a dependent instruction (henceforth termed a consumer) on the data generated by the producer can follow two policies:

a) Dispatch the instruction only when it is guaranteed that the data will be available.

b) Dispatch the instruction assuming that the producer will hit in the first level of the data cache.

Policy (b), then, dispatches the consumer instruction speculatively (a hit is always assumed for the producer instruction since the hit ratio in a cache is usually very high). If the consumer instruction arrives to the FU and the data is still not available, the instruction has to either stall at the FU or be rescheduled for dispatch in a later cycle (this option will allow other non-dependent instructions to be dispatched to the FU). In any case, both options degrade the performance of the processor.

Policy (a) provides the lowest performance since the consumer instruction might be unnecessarily stalled before it is dispatched. The producer instruction will be dispatched as soon as the producer hits in the data cache or, in case it misses, when the missing data arrives from the next level of memory hierarchy. On the other hand, this policy provides the simplest implementation, since no re-scheduling will occur.

In an embodiment of the present invention a hit/miss predictor enhances the performance of policy (b) by predicting whether the producer will hit in the data cache. Thus, the consumer instructions of a producer that is predicted to miss in the data cache will be dispatched following policy (a). If the producer instruction is predicted to hit, then the dispatch policy is (b). In this case, however, the re-scheduling logic is still needed in case the prediction is incorrect. Only in the case in which the prediction is a hit but the real outcome is a miss, the consumer instructions will need to be either stalled at the FUs or re-scheduled.

In general, the hit/miss predictor operating at the dispatch level optimizes the dispatch of consumer instructions by predicting the latency of the data. If a hit in the L1 is predicted, the latency of the data is predicted to

be the latency of the L1 cache. If a miss is predicted, the predicted latency of the data depends on whether more levels of cache exist and on whether a hit/miss predictor exists for each of these levels. If, for example, two levels of cache exist and the hit/miss outcome of the L2 is also predicted, the predicted latency of the data is computed as shown in Fig. 3 (Note: the necessary cycles, if any, to bring the data from the output of the cache to the input of the functional unit where the consumer will be executed need to be added to the predicted latency of the data).

The benefits of a hit/miss predictor for dispatch logic are not restricted to multistreaming processors only, but in a multistreaming processor where the technique has larger benefits than in a conventional (single-streaming) processor architecture. In a conventional processor having a data hit/miss predictor, when a data cache miss is predicted, no instructions (in case of an in-order dispatch engine), or only those that do not depend on the missing data (in case of an out-of-order dispatch engine) can execute. In any case, the processor resources might be idle for several cycles until the missing data is available. In multistreaming processors those idle cycles can be used to execute other instructions from other threads since they do not depend on the missing data. Thus, for a multistreaming processor, the benefits of a data cache hit/miss predictor are twofold as shown in Figure 3.

In alternative embodiments of the invention the prediction can be done differently at the fetch and dispatch stages (i.e. using different information on which to base the prediction and/or using a different prediction algorithm). As an example, the prediction at the dispatch stage could use the program counter (PC) address of the consumer instruction (since the instruction has already been decoded and its PC is known) and could follow an algorithm similar to the prediction schemes used in branch

prediction. The prediction at the fetch stage may use another type of address (cache line, for example) or other non-address information.

The prediction algorithm in different embodiments may vary depending on the workload that the processor has to efficiently support. For traditional applications, like Windows programs or SPEC benchmarks, similar algorithms to those used in branch prediction may produce the desired prediction accuracy in both hits and misses. For other types of workloads, like packet processing applications in network processors, the predictors can take advantage of additional information, like the flow number to which the packet being processed belongs (the data cache accesses performed by the processing of the first packet(s) of a new flow most likely will miss).

It will be apparent to the skilled artisan that there are many alterations that might be made in the embodiments of the invention taught herein without departing from the spirit and scope of the invention. The hit - miss predictors may be implemented in various ways, for example, and different actions may be taken based on assigned probabilities. Further, the predictors may be used at different levels in a pipeline. For example, a predictor may have input from a decode stage, and output to a fetch algorithm. Further, the mechanisms to accomplish different embodiments of the invention may be implemented typically in either hardware or software. There are similarly many other alterations that may be made within the spirit and scope of the invention. The invention should be accorded the scope of the claims below.